学习内容

- 函数式编程概念
- Lambda表达式
- Stream API

函数式编程概念

什么是函数式编程

面向对象编程是对数据进行抽象，而函数式编程是对行为进行抽象。现实世界中，数据和行为并存，程序也是如此，因此这两种编程方式我们都得学

重点：在思考问题时，使用不可变值和函数，函数对一个值进行处理，映射成另一个值

谈 Java 程序员如何定义函数式编程还为时尚早，但是，这根本不重要！我们关心的是如何写出好代码，而不是符合函数式编程风格的代码。

为什么要学函数式编程

- 用函数（行为）对数据处理，是学习大数据的基石
- 好的效率（并发执行）
- 成一个功能使用更少的代码（简介）
- **对象转向面向函数编程的思想有一定难度，需要大量的练习**

Lambda表达式

什么是Lambda表达式

Lambda是一个匿名函数，即没有函数名的函数(简化了匿名委托的使用，让你让代码更加简洁)

Lambda表达式实例

```
//匿名内部类
Runnable r = new Runnable() {
    @Override
    public void run() {
        System.out.print("hello toroot");
    }
};

//lambda
Runnable r2 = ()->System.out.print("hello toroot");

//匿名内部类
TreeSet<String> ts = new TreeSet<>(new Comparator<String>() {
    @Override
    public int compare(String o1, String o2) {
        return Long.compare(o1.length(),o2.length());
    }
});

```

```
//lambda
TreeSet<String> ts2 = new TreeSet<>((o1,o2)->
Long.compare(o1.length(),o2.length()));
```

Lambda 表达式语法

Lambda 表达式在Java 语言中引入了一个新的语法元素和操作符。这个操作符为 “->”， 该操作符被称为 Lambda 操作符或剪头操作符。

它将 Lambda 分为两个部分：

a. 左侧：指定了 Lambda 表达式需要的所有参数
b. 右侧：指定了 Lambda 体，即 Lambda 表达式要执行的功能。

Lambda 表达式语法格式

- 一、语法格式一：无参数，无返回值
  `() -> System.out.println("Hello Lambda!");`
- 二、 语法格式二：有一个参数，并且无返回值
  `(x) -> System.out.println(x)`
- 三、 语法格式三：若只有一个参数，小括号可以省略不写
  `x -> System.out.println(x)`
- 四、 语法格式四：有两个以上的参数，有返回值，并且 Lambda 体中有多条语句

```
        Comparator<Integer> com = (x, y) -> {
            System.out.println("函数式接口");
            return Integer.compare(x, y);
        };
```

- 五、 语法格式五：若 Lambda 体中只有一条语句， return 和 大括号都可以省略不写
  `Comparator<Integer> com = (x, y) -> Integer.compare(x, y);`
- 六、 语法格式六：Lambda 表达式的参数列表的数据类型可以省略不写，因为JVM编译器通过上下文推断出，数据类型，即“类型推断”
  `(Integer x, Integer y) -> Integer.compare(x, y);`

习题：

- 定义一个接口，接收两参数（两个参数类型一样的泛形），返回int
- 定义一个person类，3个属性name,age,score

作业：

1. 实现接口，比较两个的age大小
2. 实现接口，比较两个的score大小
3. 使用内置的compare接口对数组排序

函数式接口

Lambda 表达式需要“函数式接口”的支持
函数式接口：接口中只有一个抽象方法的接口，称为函数式接口。 可以使用注解
`@FunctionalInterface` 修饰可以检查是否是函数式接口

```
@FunctionalInterface
public interface MyFun {
```

```
    public double getValue();
}

@FunctionalInterface
public interface MyFun<T> {
    public T getValue(T t);
}

public static void main(String[] args) {
    String newStr = toUpperString((str)->str.toUpperCase(),"toroot");
    System.out.println(newStr);
}

public static String  toUpperString(MyFun<String> mf,String str) {
    return mf.getValue(str);
}
```

Java内置函数式接口

接口	参数	返回类型	示例
Predicate	T	boolean	这帅哥是源本学院
Consumer	T	void	输出一个值
Function<T,R>	T	R	获得 Person对象的名字
Supplier	None	T	工厂方法
UnaryOperator	T	T	逻辑非 (!)
BinaryOperator	(T, T)	T	求两个数的乘积 (*)

练习题:

- 使用Predicate接口，判断某个person对象是否大于18岁
- 使用Consumer接口，传入一个字符串，打印字符串长度
- 使用Function接口，传入一个字符串，返回字符串长度
- 使用Supplier接口，创建一个姓名等于张三，年龄28，成绩81的person对象
- 使用UnaryOperator接口，传入一个字符串，返回全转大写的新字符串

方法引用

- 非重点，但得看得懂
  当要传递给Lambda体的操作，已经有实现的方法了，可以使用方法引用！（实现抽象方法的参数列表，必须与方法引用方法的参数列表保持一致！）
  方法引用：使用**操作符 “ ::” 将方法名和对象或类的名字分隔开来。**
  如下三种主要使用情况：
- 对象 :: 实例方法
- 类 :: 静态方法
- 类 :: 实例方法

什么时候可以用 ::方法引用（重点）

在我们使用Lambda表达式的时候，”->”右边部分是要执行的代码，即要完成的功能，可以把这部分称作Lambda体。有时候，当我们想要实现一个函数式接口的那个抽象方法，但是已经有类实现了我们想要的功能，这个时候我们就可以用方法引用来直接使用现有类的功能去实现

文字解释有点绕，我们直接上代码

```
 Person p1 = new Person("Av",18,90);
        Person p2 = new Person("King",20,0);
        Person p3 = new Person("Lance",17,100);
        List<Person> list = new ArrayList<>();
        list.add(p1);
        list.add(p2);
        list.add(p3);

        //这里我们需要比较list里面的person,按照年龄排序
        //那么我们最常见的做法是
        //sort(List<T> list, Comparator<? super T> c)
        //1. 因为我们的sort方法的第二个参数是一个接口，所以我们需要实现一个匿名内部类
        Collections.sort(list, new Comparator<Person>() {
            @Override
            public int compare(Person person1, Person person2) {
                return person1.getAge().compareTo(person2.getAge());
            }
        });
        //2. 因为第二个参数是一个@FunctionalInterface的函数式接口，所以我们可以用lambda写法
        Collections.sort(list, (person1,person2) -> p1.getScore().compareTo(p2.getAge()));
        //3. 因为第二个参数我们可以用lambda的方式去实现，
        // 但是刚好又有代码(Comparator.comparing)已经实现了这个功能
        // 这个时候我们就可以采用方法引用了
        /**
         * 重点：
         * 当我们想要实现一个函数式接口的那个抽象方法，但是已经有类实现了我们想要的功能，
         * 这个时候我们就可以用方法引用来直接使用现有类的功能去实现。
         */
        Collections.sort(list, Comparator.comparing(Person::getAge));

        System.out.println(list);
```

```
public static void main(String[] args) {
    Consumer<String>  c = x->System.out.println(x);
    //等同于
    Consumer<String> c2 = System.out::print;
}

public static void main(String[] args) {
    BinaryOperator<Double> bo = (n1,n2) ->Math.pow(n1,n2);
    BinaryOperator<Double> bo2 = Math::pow;
}

public static void main(String[] args) {
    BiPredicate<String,String> bp = (str1,str2) ->str1.equals(str2);
    BiPredicate<String,String> bp2 = String::equals;
}
```

注意：
当需要引用方法的第一个参数是调用对象，并且第二个参数是需要引用方法的第二个 参数((或无参数)) 时： ClassName::methodName

构造器引用（非重点，但得看得懂）

格式： ClassName :: new
与函数式接口相结合，自动与函数式接口中方法兼容。
可以把构造器引用赋值给定义的方法，与构造器参数列表要与接口中抽象方法的参数列表一致！

```
public static void main(String[] args) {
    Supplier<Person> x = ()->new Person();
    Supplier<Person> x2 = Person::new;
}

public static void main(String[] args) {
    Function<String,Person> f  = x->new Person(x);
    Function<String,Person> f2 = Person::new;
}
```

数组引用

非重点，但得看得懂
格式： type[] :: new

```
public static void main(String[] args) {
    Function<Integer,Person[]> f  = x->new Person[x];
   Function<Integer,Person[]>  f2 = Person[]::new;
}
```

Stream API

流 (Stream)是什么

Stream是数据渠道，用于操作数据源（集合、数组等）所生成的元素序列。
“集合讲的是数据，流讲的是计算！ ”
注意：

- Stream 自己不会存储元素。
- Stream 不会改变源对象。相反，他们会返回一个持有结果的新Stream。
- Stream 操作是延迟执行的。这意味着他们会等到需要结果的时候才执行。

Stream 操作实例

取出所有大于18岁人的姓名，按字典排序，并输出到控制台

```
    private static  List<Person> persons = Arrays.asList(
             new Person("CJK",19,"女"),
             new Person("BODUO",20,"女"),
             new Person("JZ",21,"女"),
             new Person("anglebabby",18,"女"),
             new Person("huangxiaoming",5,"男"),
             new Person("ROY",18,"男")
    );
public static void main(String[] args) throws IOException {
 persons.stream().filter(x->x.getAge()>=18).map(Person::getName).sorted().forEach(System.out::println);
}
```

Stream 的操作三个步骤

1. 创建 Stream
   一个数据源（如：集合、数组），获取一个流
2. 中间操作
   一个中间操作链，对数据源的数据进行处理
3. 终止操作(终端操作)
   一个终止操作，执行中间操作链，并产生结果

创建Steam

- Collection 提供了两个方法 stream() 与 parallelStream()
- 通过 Arrays 中的 stream() 获取一个数组流
- 通过 Stream 类中静态方法 of()
- 创建无限流

1. 创建 Stream

```
@Test
public void test1(){
    //1. Collection 提供了两个方法  stream() 与 parallelStream()
    List<String> list = new ArrayList<>();
    Stream<String> stream = list.stream(); //获取一个顺序流
    Stream<String> parallelStream = list.parallelStream(); //获取一个并行流

    //2. 通过 Arrays 中的 stream() 获取一个数组流
    Integer[] nums = new Integer[10];
    Stream<Integer> stream1 = Arrays.*stream*(nums);

    //3. 通过 Stream 类中静态方法 of()
    Stream<Integer> stream2 = Stream.of(1,2,3,4,5,6);

    //4. 创建无限流
    //迭代
    Stream<Integer> stream3 = Stream.iterate(0, (x) -> x + 2).limit(10);
    stream3.forEach(System.out::println);

    //生成
    Stream<Double> stream4 = Stream.generate(Math::random).limit(2);
    stream4.forEach(System.out::println);
}
```

问题：
下面创建流(Stream)的方式哪些是正确的（多选）
A. Steam.newInstanceOf()
B. Collection.of()
C. Collection.stream() 或Collection.parallelStream()
D.Stream.of()
E.Stream.generate() 或 Stream.iterate()
F.Arrays.stream()

中间操作

1. 筛选与切片
    - filter——接收 Lambda ， 从流中排除某些元素。
    - limit——截断流，使其元素不超过给定数量。
    - skip(n) —— 跳过元素，返回一个扔掉了前 n 个元素的流。若流中元素不足 n 个，则返回一个空流。与 limit(n) 互补
    - distinct——筛选，通过流所生成元素的 hashCode() 和 equals() 去除重复元素
2. 映射

- map——接收 Lambda ， 将元素转换成其他形式或提取信息。接收一个函数作为参数，该函数会被应用到每个元素上，并将其映射成一个新的元素。
- flatMap——接收一个函数作为参数，将流中的每个值都换成另一个流，然后把所有流连接成一个流

3. 排序

- sorted()——自然排序
- sorted(Comparator com)——定制排序
  问题：

1. 有个数组Integer[] ary = {1,2,3,4,5,6,7,8,9,10}，取出中间的第三到第五个元素

```
List<Integer> collect =
Arrays.stream(ary).skip(2).limit(3).collect(Collectors.toList());
```

2. 有个数组Integer[] ary = {1,2,2,3,4,5,6,6,7,8,8,9,10}，取出里面的偶数，并去除重复

```
List<Integer> list = Arrays.stream(ary).filter(x -> x % 2 ==
0).distinct().collect(Collectors.toList());
Set<Integer> integerSet = Arrays.stream(ary).filter(x -> x % 2 ==
0).collect(Collectors.toSet());
```

3. 有个二维数组，要求把数组组合成一个一维数组，并排序（1，2，3，4，5.....12）
   Integer[][] ary = {{3,8,4,7,5}, {9,1,6,2}, {0,10,12,11} };

```
Arrays.stream(ary).flatMap(item-
>Arrays.stream(item)).sorted().forEach(System.out::println);
```

终止操作

终端操作会从流的流水线生成结果。其结果可以是任何不是流的值，例如：List、Integer，甚至是 void 。

1. 查找与匹配

接口	说明
allMatch(Predicate p)	检查是否匹配所有元素
anyMatch(Predicate p)	检查是否至少匹配一个元素
noneMatch(Predicate p)	检查是否没有匹配所有元素
findFirst()	返回第一个元素
findAny()	返回当前流中的任意元素
count()	返回流中元素总数
max(Comparator c)	返回流中最大值
min(Comparator c)	返回流中最小值
forEach(Consumer c)	迭代

问题：
Integer[] ary = {1,2,3,4,5,6,7,8,9,10}

1. 检查是否所有元素都小于10
2. 检查是否至少有一个元素小于2
3. 检查是不是没一个元素大于10
4. 返回第一个元素
5. ary 有多少个元素
6. 求ary里面最大值
7. 求ary里面最小值
8. 循环遍历打出ary 里面偶数

归约

reduce(T iden, BinaryOperator b) 可以将流中元素反复结合起来，得到一个值。返回 T
reduce(BinaryOperator b) 可以将流中元素反复结合起来，得到一个值。返回 Optional

问题：求所有人员学生的总分

```
   Integer all = persons.stream().map(Person::getScore).reduce((integer,
integer2) -> integer + integer2).get()
```

收集

collect(Collector c)
将流转换为其他形式。接收一个 Collector接口的实现，用于给Stream中元素做汇总的方法
Collector 接口中方法的实现决定了如何对流执行收集操作(如收集到 List、Set、Map)。
Collectors实用类提供了很多静态方法，可以方便地创建常见收集器实例
具体方法与实例如下：

- toList List 把流中元素收集到List

```
List<Person> emps= list.stream().collect(Collectors.toList());
```

- toSet Set 把流中元素收集到Set

```
Set<Person> emps= list.stream().collect(Collectors.toSet());
```

- toCollection Collection 把流中元素收集到创建的集合

```
Collection<Person>
emps=list.stream().collect(Collectors.toCollection(ArrayList::new));
```

- counting Long 计算流中元素的个数

```
long count = list.stream().collect(Collectors.counting());
```

- summing Int Integer 对流中元素的整数属性求和

```
int total=list.stream().collect(Collectors.summingInt(Person::getAge));
```

- averaging Int Double 计算流中元素Integer属性的平均值

```
double avg= list.stream().collect(Collectors.averagingInt(Person::getAge));
```

- summarizingInt IntSummaryStatistics 收集流中Integer属性的统计值。如：平均值

```
Int SummaryStatisticsiss=
list.stream().collect(Collectors.summarizingInt(Person::getAge));
```

- joining String 连接流中每个字符串

```
String str= list.stream().map(Person::getName).collect(Collectors.joining());
```

- maxBy Optional 根据比较器选择最大值

```
Optional<Person> max=
list.stream().collect(Collectors.maxBy(comparingInt(Person::getSalary)));
```

- minBy Optional 根据比较器选择最小值

```
Optional<Person> min =
list.stream().collect(Collectors.minBy(comparingInt(Person::getSalary)));
```

- reducing 归约产生的类型 从一个作为累加器的初始值开始，利用BinaryOperator与流中元素逐个结合，从而归约成单个值

```
int total=list.stream().collect(Collectors.reducing(0, Person::getSalar,
Integer::sum));
```

- collectingAndThen 转换函数返回的类型 包裹另一个收集器，对其结果转换函数

```
int how=
list.stream().collect(Collectors.collectingAndThen(Collectors.toList(),
List::size));
```

- groupingBy Map<K, List> 根据某属性值对流分组，属性为K，结果为V

```
Map<Person.Status,List<Person>> map=
list.stream().collect(Collectors.groupingBy(Person::getStatus));

partitioningBy Map<Boolean, List<T>> 根据true或false进行分区

Map<Boolean,List<Person>>vd=
list.stream().collect(Collectors.partitioningBy(Person::getManage));
```

练习
Integer[] ary = {1,2,3,4,5,6,7,8,9,10}

1. 使用Collectors求ary的最大值
2. 使用Collectors求ary的平均值
3. 使用Collectors.joining输出”1:2:3:4:5:6:7:8:9:10”
4. 使用Collectors.reducing求ary数组的总和
5. 使用Collectors.counting求ary个数

问题：

1. 取出Person对象的所有名字，放到List集合中

```
List<String> collect2 =
persons.stream().map(Person::getName).collect(Collectors.toList());
```

2. 求Person对象集合的分数的平均分、总分、最高分，最低分，分数的个数

```
IntSummaryStatistics collect =
persons.stream().collect(Collectors.summarizingInt(Person::getScore));
System.out.println(collect);
```

3. 根据成绩分组，及格的放一组，不及格的放另外一组

```
Map<Boolean, List<Person>> collect1 =
persons.stream().collect(Collectors.partitioningBy(person ->
person.getScore() >= 60));
System.out.println(collect1);
```

4. WordCount

```
public static void main(String[] args) throws IOException {
    InputStream resourceAsStream =
Person.class.getClassLoader().getResourceAsStream("aa.txt");
    BufferedReader bufferedReader = new BufferedReader(new
InputStreamReader(resourceAsStream));
    bufferedReader.lines().flatMap(x->Stream.of(x.split("
"))).sorted().collect(Collectors.groupingBy(String::toString)).forEach((a,b)
-> System.out.println(a+":"+b.size()));
    bufferedReader.close();
}
```